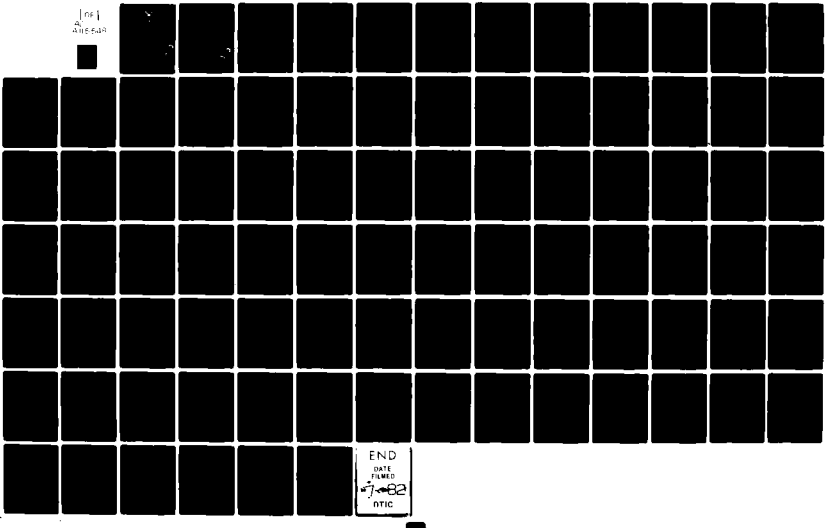
AD-A115 548 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
CONTINUED DEVELOPMENT OF THE JOVIAL(J73) TO ADA TRANSLATOR SYST--ETC(U)
DEC 81 E L EAST
UNCLASSIFIED AFIT/GCS/MA/81D-2 NL



UNITED STATES AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

CONTINUED DEVELOPMENT OF THE JOVIAL(J73) TO ADA TRANSLATOR SYSTEM

THESIS

AFIT/GCS/MA/81D-2
Eric L. East
Capt USAF

CONTINUED DEVELOPMENT OF THE JOVIAL(J73) TO ADA TRANSLATOR SYSTEM

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air Training Command

in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

by

Eric L. East, B.S.

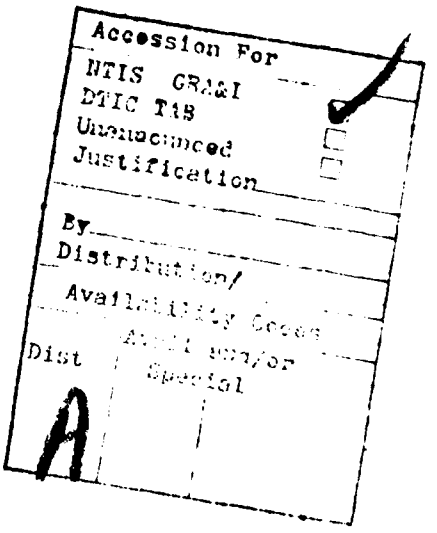
Capt USAF

Graduate Computer Systems

December 1981

Table of Contents

List of Figures



Preface

I became interested in digital computers several years ago and have since tried to fully exploit their power to accomplish repetitious jobs. Later, while working on my undergraduate degree, I became especially interested in programming languages and particularly in compilers. I chose to pursue the continued development of the JOVIAL(J73) to Ada Translator System (JATS) begun by Capt Richard L. Brozovic (Ref 1) so I could apply some of the parsing techniques studied in courses on compiler theory and construction at the Air Force Institute of Technology. This project also afforded me the opportunity to examine some of the current works in the area of language translation, to study Ada, and to learn JOVIAL(J73) since the translator system is written in J73. Using a computer to perform language translation is a very good way to exploit its power.

The Department of Defense (DoD) has selected Ada as the new standard high order programming language to replace JOVIAL(J73), the current standard, for programming embedded-computer systems. Since the Air Force has a considerable investment in software using J73, the need for a J73-to-Ada translator is very real. The sponsor, the Avionics Laboratory of the Air Force Wright Aeronautical Laboratories (AFWAL/AA), is very interested in this project. In addition, Headquarters, Air Force Systems Command, Andrews AFB, Maryland, the Satellite Control Facility, Sunnyvale AFS,

California, and Rome Air Development Center, Griffiss AFB, New York have expressed a great interest in this project.

Several persons deserve my thanks for their help and advice given during the life of this project. Major Caro (AFWAL/AA) was the project sponsor and provided guidance and advice during the initial stages of getting the original translator running. Mr. Mike Burlakoff was extremely helpful in getting the translator running under the new version of the J73 compiler. He also provided answers to the detailed problems I encountered in using and understanding J73. Mr. Bob Engimann, TRW Inc., deserves special thanks in the area of the J73 compiler. He provided indispensable information in regard to generated code for suspected compiler problems and also in the area of error message interpretation. 1Lt Bob Sudduth's help with J73 problems was greatly appreciated. I also obtained help from many others from time to time. I wish to express my gratitude to AFWAL/AA for their sponsorship and support of this project.

I want to thank my thesis advisor, Capt Roie Black, for his guidance and occasional prodding during this thesis effort.

Eric L. East

Abstract

JOVIAL(J73) is the current Air Force standard (MIL-STD-1589B)(Ref 2) high order programming language for embedded-computer systems. The Department of Defense (DoD) recently announced Ada as the new standard high order language for embedded-computer software which will replace J73. The overall Ada project consists of two distinct parts: a programming language and the Ada Programming Support Environment (APSE) which is being defined in conjunction with the language. One feature of this environment is a toolset, which contains software tools written in Ada which are necessary for the development and maintenance of Ada programs. Since the toolset should remain open-ended and new tools added as necessary, one tool that should be considered is a translator to produce Ada programs from J73 programs (Ref 3:1,24).

The aim of this thesis effort is to continue the development of the JOVIAL(J73) to Ada Translator System (JATS) originally developed as a thesis project by Capt Richard L. Brozovic. The current translator processes a subset of J73 using a bottom-up table driven LR(1) parser and builds a parse tree. The J73 parse tree is then transformed into an equivalent Ada parse tree. Finally a pretty-print procedure processes the Ada tree and generates the new Ada source program with the proper indentation.

The translator system is currently hosted on a Digital Equipment Corporation (DEC-10) computer system at the Avionics Laboratory of the Wright Aeronautical Laboratories (AFWAL/AA), Wright-Patterson AFB, Ohio. Although the translator is written in standard JOVIAL(J73), it is not directly rehostable on other computer systems due to the locally standard FORTRAN implemented I/O interface used, since J73 does not have I/O .

1. Introduction

The Department of Defense (DoD) was spending approximately $3 billion per year on computer software in 1975 with prices continuing to soar. Each of the armed services has its own "so called" standard programming language which is somewhat unique to particular application. The Army has TACPOL; the Navy has CMS-2 and SPL/1; the Air Force has JOVIAL (J3 and J73). DoD conceived the idea of a single high order programming language, Ada, to be the new standard used throughout the services for developing embedded-computer software. "'Embedded-computer systems' are those computer systems contained in larger systems; for example, a spacecraft that contains a computer which monitors navigational and flight control actions" (Ref 4:146-147). DoD's aim was to have a single powerful language thus reducing much redundancy and ultimately the soaring costs of software.

The problem that the services faced was what to do with the existing software: rewrite or translate. Since the function of the existing software is still valid, the decision should be to convert into the new common language, either manually or automatically. The idea of automatic conversion (translation) has much more appeal than does manual conversion.

The overall Ada project consists of a programming

language plus the Ada Programming Support Environment (APSE). One of the primary features of the APSE is the toolset. This toolset contains programs written in Ada which are necessary for the development and maintenance of Ada software. Since Ada is destined to replace Air Force JOVIAL(J73) for programming embedded-computer systems and since the Air Force has a considerable investment in J73 software, the need for a J73-to-Ada translator is very real. Although direct translation of existing programs may not produce production quality software, a translator that performs the bulk of the translation could still be a valuable aid in the transition to the new programming language. It should save valuable time by reducing the amount of hand translation that is required while identifying those areas that do require manual translation. Capt Richard L. Brozovic developed the skeleton of the J73-to-Ada translator system, known as JATS, which serves as the basis for continuing development (Ref 1).

Since this is a continued development effort, it is important to study the existing translation techniques in order to establish a basic understanding of these methods and be able to apply them to the translator system when applicable. A brief survey of current language translation techniques will be presented.

1.1 Review of Existing Translation Techniques

Almost all the work that has been done and is being done in the area of language translation employ established

methods of recursive definition for the language's grammar. The particular language's syntax has its grammar defined recursively using a standard technique known as Backus-Naur Form (BNF). BNF consists of a set of production rules that contain terminal and nonterminal symbols. Refer to Section 1.1.1 for a fuller explanation of production rules as they pertain to parsing a string in some grammar. The parsing is usually accomplished using a table driven bottom-up LR(1) parser. A parser is some mechanism capable of determining if a construct (string) belongs to a particular grammar (language). Parsers are usually contained in two broad classes: top-down and bottom-up (Ref 5:46-47). The "LR" means Left-to-right, Right-most and refers to the order in which the terminal nodes are read in order to form a valid string in the given grammar. The "1" indicates the number of lookahead or additional symbols in the input stream beyond the current one that must be examined to determine the appropriate production choice (Ref 5:60-61). A top-down recursive descent parser could be used provided the grammar is sufficiently small.

A recursive descent parser represents one of the most popular methods for parsing. This is due primarily to the method used to accomplish the parsing. The parsing and semantic operations are contained in a single computer program and consists of recursive procedure calls which are understood by most programmers. The basic idea is that every nonterminal is connected to a recursive procedure call.

These procedures test the alternative right hand sides of individual grammar productions (Ref 5:161-162). Although the recursive descent parser is more popular, table driven parsers were typically used when the grammars were quite large. Another important point to keep in mind is that the recursive descent is very much language dependent. A minor grammar change could mean a complete rewrite of the program. On the other hand, the table driven parser is a more general parser where tables contain the necessary information to parse the particular grammar. The program that does the parsing simply performs a series of table lookups and only the tables need be changed when the grammar changes.

1.1.1 Example Grammar

A small example arithmetic expression grammar, denoted as G, will be presented in order to explain the concept of terminals, nonterminals, productions, and how parsing is done. Refer to Figure 1-1. This context-free grammar was taken directly from Barrett and Couch (Ref 5:22). Context-free grammars constitute a broad class of grammars.

Some definitions are needed before we proceed. A nonterminal is some larger piece of the grammar. For example, F in the production F --> (E) is a nonterminal that represents some larger parenthesized expression in the grammar G. Any nonterminal symbol may be replaced by the right side of the production when appropriate. For example, T may be replaced by T * F as shown in Figures 1-1 and 1-2.

A terminal represents an indivisible element in the grammar. The parentheses in the above production are terminals in the grammar as will be noted later.

Production Rules for G

1. E --> E + T
2. E --> T
3. T --> T * F
4. T --> F
5. F --> (E)
6. F --> a

Figure 1-1: Example Arithmetic Expression Grammar: G

The nonterminals in G include E, T, and F. Nonterminals are often written with brackets around them (i.e. <E>) to help separate them from the terminals in the productions, but in this simple example the brackets will not be used. The terminals in G are the commonly recognized arithmetic operators +, *, (,), and the a which may represent some number. The parsing start symbol is E. A production is simply a rule that specifies how strings may be constructed in a given grammar. This grammar belongs to the broad class known as context-free since the left side of the production rules contain only a single nonterminal. If the left side contains terminals, then it would be classified as the other broad class of grammars known as context-sensitive.

An example string belonging to G and an explanation of how it is parsed will be presented. Let a*a be a string in G. Recall that a parser's function is to determine if a

string belongs to the given grammar as defined by the production rules. The parsing starts with the start symbol and continues until the string is derived and thus belongs to the grammar. If the string cannot be derived from the production rules, the string does not belong to the grammar. The arrow (-->) can be read as "derives". Refer to Figure 1-2.

Production Rule Applied	Meaning
1	E derives E + T (but wrong choice)
2	E derives T (choice ok)
3	T derives T * F
4	T * F derives F * F
6	F * F derives a * F
6	a * F derives a * a (string is derivable)

Figure 1-2: Parsing the String a*a

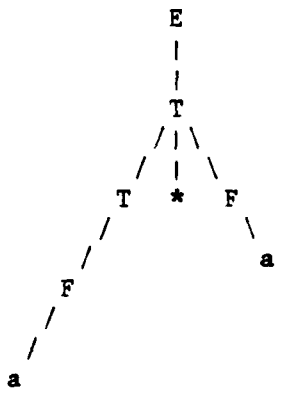


Figure 1-3: Syntax Tree for String a*a in G

Figure 1-2 shows how the given string was parsed and Figure 1-3 shows the resulting syntax tree, also referred to

as a parse tree. If the terminal nodes (leaves) are read in a left-to-right fashion, it should be obvious that the resulting string is a*a. This demonstrates that the string does, in fact, belong to G.

Note that the requirement for lookahead symbols is vividly apparent in the attempt to begin parsing with the wrong production choice (E --> E + T). The current symbol in the input stream is the first a and the additional symbol (lookahead) required to make the correct production choice is the *. Without the one symbol lookahead, this parser would not have been able to determine the correct production choice. Since a brief explanation of terms and a demonstration of how a string is parsed has been accomplished, a summary of the currently used language translation techniques will be presented.

1.1.2 A Formal System

Donovan and Ledgard (Ref 6) described a formal system for specifying the syntax and subsequent translation of a programming language. They presented two very similar recursive methods for specifying the syntax of a subset of PL/1, a declarative, block structured language: BNF and a canonic system. "A canonic system is a finite sequence of rules for recursively defining sets" (Ref 6:554). Their method was not source or target language dependent. It used an easily understood notation, and defined sets of strings recursively. The syntax of the language featured a set where

each member of the set was a syntactically correct program. The translation of a given language was marked by defining a set of ordered pairs where the first element in each pair represented the correct program in the source language and the second element represented the same correct program but in the target language. The semantics (meanings) of the source language were transferred to the target language. Once the target language was fully understood, the semantics of the source language were established (Ref 6:554).

This paper was developed around a subset of PL/1 and its translation into IBM 360 assembler language.

1.1.3 Dynamic Syntax Specification

Ginsburg and Rounds (Ref 7) described a method of syntax specification very similar to that described by Donovan and Ledgard. They also used a subset of PL/1. The grammar for the syntax specification was defined using BNF. The semantics of the grammar were constructed dynamically during the lexical analysis of the input program stream. Next, the grammar was mapped into a context-free grammar that satisfied the production rules of a given programming language (Ref 7).

1.1.4 RATFOR-FORTRAN Translator

Kernigan and Plauger presented a translator which transformed structured-FORTRAN with such control flow structures as **if-else**, **while**, **for**, and **repeat-until** into standard FORTRAN with goto's (Ref 8). The RATFOR grammar was again specified in BNF, as were the others. The typical

processing sequence for a RATFOR program is:

```
RATFOR source
--->RATFOR translator
    --->FORTRAN source
        --->FORTRAN compiler
            --->Executable FORTRAN object
                module (Ref 8:10)
```

1.1.5 Generalized Translation

The Linguistics Research Center (LRC) of The University of Texas has developed a generalized model for the translation of both computer languages as well as natural languages (Ref 9). The LRC used the terminology of the linguistic theory of transformational grammar to describe the model. The theory states that a language has two structures rather than only one: a basic structure and a surface structure. The model consists of a base element that creates basic trees while the transformational element maps the basic trees into a surface tree. The leaves (terminal nodes) of the surface tree represent strings defined by a particular language.

One could view these basic and surface trees as not being structures of only a single language but structures of two different languages: a source language and a target language, respectively. This may help clarify the idea of translating one language (source) into another (target) by applying this technique in a little different way.

The base element is considered phase one of the translation process. Although the basic trees have terminal nodes (leaves), they do not necessarily represent a meaningful string in the target language. However, in the final phase of the translation process, the transformational element is expressed as a grammar with specific ordering rules for the target string formats. These rules are applied to the basic tree and the leaves are placed in the proper order. These newly arranged leaves must form a string, reading from left-to-right, which belongs to the target language.

One must make a transformational mapping from a source language to a target language if the source string is not valid for the target language. For example, let Figure 1-4 represent a basic tree in some language X. If the resulting string abcedf does not represent a valid string in the target language Y, then the string must be transformed to yield a valid string in Y. Figure 1-5 shows that the terminal branch d is moved from interior node C to B and abcdef now represents a valid string in Y (Ref 9:570). Brozovic (Ref 1) used this technique to transform a J73 string into an equivalent Ada string as will be seen.

In the example represented by Figures 1-4 and 1-5 the terminals are represented by a, b, c, e, d, and f while the interior nodes A, B, and C represent the nonterminals. Terminals only appear at the end nodes (leaves) on a syntax

or parse tree.

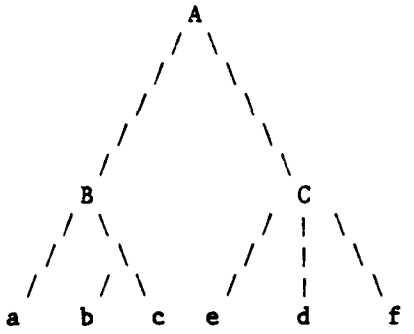


Figure 1-4: Basic tree

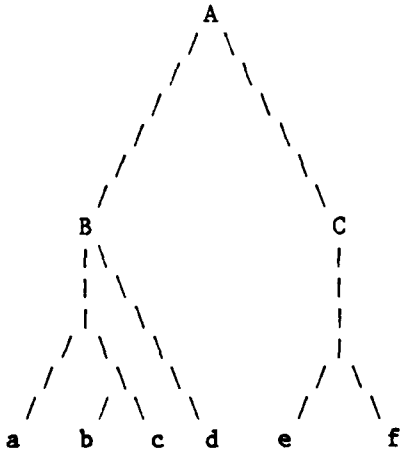


Figure 1-5: Surface tree

Jonas (Ref 9:576) pointed out that there still exists a problem in mapping primitives of one language onto those of another. He further stated that only when the source and target languages are very much alike will there be primitives common to both. It should be noted that the Ada and J73 languages are very similar and have many common primitives.

An example where few common primitives exist is the case where an assembler language is translated into a high order language. Sassaman (Ref 10) described such a case in the translation of machine code into FORTRAN.

1.2 Purpose

The purpose of this thesis project is to continue the development of the J73-to-Ada translator (Ref 1). The major thrust is to get the original translator system up-to-date in the current DEC-10 environment so that the remaining J73 syntax can be translated, thereby providing the Avionics Laboratory with a useful and useable software product. The translator system will continue to be refered to as JATS and will remain hosted on the AFWAL/AA's DEC-10 computer system.

1.2.1 Scope

Since the original translator was designed to translate only a subset of J73 as defined by the outdated MIL-STD-1589A, deficiencies in regard to the complete syntax translation, probably exists in the original translator system. Therefore, the translator was studied in detail, and all discovered or suspected deficiencies were properly documented. Modifications were accomplished as needed. The overall objective of this effort was to develop a step-by-step framework so further development efforts could proceed more smoothly to allow most of the complete J73 syntax to be translated. It is hoped that this framework will help speed the continued development and eventual deployment

of the translator system since Ada compilers are soon to emerge.

1.2.2 Assumptions

It was assumed that Brozovic's skeleton translator was valid and worked without modifications to perform the translation of the subset of J73 syntax as stated in his thesis (Ref 1). Since the translator does not perform any syntax or error checking, it was assumed that all input J73 programs were syntactically correct.

1.2.3 Standards

The continued development of the translator will be governed by MIL-STD-1589B (Ref 2) and the DoD REFERENCE MANUAL FOR THE ADA PROGRAMMING LANGUAGE (Ref 11).

1.2.4 Target Environment

The Avionics Laboratory agreed to provide all necessary computer support on the DEC-10 and other related equipment to continue the translator development.

1.3 Contents

The remainder of this report covers the various aspects of the continued development efforts of JATS. Chapter 2 covers briefly the requirements definition as presented by AFWAL/AA. Chapter 3 presents a detailed critical analysis of the original translator system. Chapter 4 describes the software development efforts necessary to get JATS up-to-date in its present environment and to stablize the software.

Chapter 5 contains the results and discussions of the thesis project. Chapter 6 contains the recommendations for further refinements of JATS. Appendix A contains the JATS user's guide and the maintenance programmer's guide which contains detailed information necessary for the continued development of the translator. Appendix B presents an analysis of the run-time processes of the translator system. Appendix C contains a summary of the major modifications made to JATS, as well as, a description of the new procedures added to JATS. Appendix D provides a complete list of the J73 grammar terminals (tokens) and nonterminals. Appendix E contains a test J73 program used for translation and Appendix F contains the equivalent Ada version of the J73 program with the untranslated source code identified.

2. Requirements Definition

The requirements as stated in Brozovic's original work are still valid (Ref 1:5-6). Not only is the Avionics Laboratory very interested in a J73-to-Ada translator but other organizations have expressed interest as was pointed out earlier.

Just as the introduction of Ada is designed to help reduce future software costs within DoD, it is appropriate to build upon someone else's work to get a more complete and useful product in a given amount of time. The translator will continue to be developed in a modular fashion thus allowing future development efforts to progress without disturbing the surrounding source code. The first priority is to bring it up-to-date. The second priority is to increase the reliability of the software by conducting a detailed analysis, completing the implementation of the algorithm, performing testing, and adding additional in-line documentation. The last priority is to complete the remaining J73 translation. There would be time later to fine tune it or make it more flexible.

A brief summary of the Avionics Laboratory's requirements are:

1. The translator should accept syntactically correct J73 source programs and produce equivalent Ada source programs.
2. If translation is not possible, then the input

source shall be identified to allow for manual translation.

3. The source program shall conform to MIL-STD-1589B, the JOVIAL (J73) standard, to allow maximum portability. Refer to Section 5.8 for the portability discussion.

4. A final and essential requirement is to add additional in-line documentation and provide a more detailed JATS user's/maintenance programmer's guide. AFWAL/AA expressed a strong desire in this area because such documentation would be vital if further work on JATS is to be continued without too much confusion (Ref 1:5-6).

JATS shall remain open-ended to allow future work to be continued with a minimum of impact on earlier efforts. Since the complete J73 syntax will not be translated during this current effort, the remaining parts can be added easily at a future date.

3. Analysis of the Original Translator System

A complete analysis of any large software project must be performed whether it is an initial software development or a continuation of a previous effort. A good analysis should save much rewriting and redesign later because of false starts. An analysis of the existing software is essential for a complete understanding of its functions.

The familiarization phase of any enhancement to an existing software project is a time of guessing at functions performed, wondering why this or that was done, and gaining as much understanding of the software as best one can. This phase was essential before any modifications could be done correctly.

In the remainder of this report, for clarity, reserved words and specific data names (terminals) will be capitalized to distinguish them from surrounding text. Nonterminals in J73 will be enclosed in angle brackets <...>. For instance <name> would refer to any valid J73 symbol or identifier. Throughout this report the terms **tokens** and **terminals** will be used synonymously. The elements of the language conform as much as possible to Military Standard JOVIAL (J73) (MIL-STD-1589B) (Ref 2).

It should be noted that JATS also employs many of the established techniques as summarized in Sections 1.1. These similarities are enumerated below.

1. The J73 grammar is defined in BNF.

2. JATS performs the parsing using table driven bottom-up techniques.

3. JATS processing sequence follows essentially the same sequence as did the RATFOR translator shown in Section 1.1.4.

```
J73 source

--->J73 translator (JATS)

    --->Ada source

        --->Ada compiler

            --->Executable Ada object module
```

4. Brozovic's translator (Ref 1) utilizes the same technique to translate a J73 string into an equivalent Ada string as discussed in Section 1.1.5. If terminal nodes are out of order, they are rearranged in the proper order. If the source language nodes have no equivalency in the target language, they are simply deleted. By the same token, if the target language has nodes that have no equivalency in the source language, the missing nodes are added to the surface tree. The result of this transformational process is two strings with equivalent semantics but possibly different structures. In regard to JATS , the basic tree or source tree represents an element of JOVIAL(J73) and the surface tree or target tree represents the equivalent element in Ada.

3.1 Example J73 <case> Translation

A J73 <case> construct will be used to demonstrate how the translator system actually performs the J73-to-Ada translation. Figure 3-1 shows a J73 <case> construct and Figure 3-2 shows the equivalent structure in Ada. Since there are quite a large number of J73 productions involved in parsing the case statement, assume for sake of brevity, that the statement has been successfully parsed and the

corresponding partial J73 parse (syntax) tree shown in Figure 3-3 has been generated. The contents of the interior nodes are of no value in this discussion and are not shown. The parsing of the case statement would have been accomplished in the same manner as was shown earlier for the example string a*a in grammar G. Refer to Section 1.1. The parsing would have been controlled by using the appropriate J73 production rules.

```
CASE NUM ;
BEGIN "case"

(DEFAULT) : S'1;
(1)       : S'2;

END   "case"
```

Figure 3-1: J73 Case Construct

```
CASE NUM IS
BEGIN --case

WHEN 1      => S_2;
WHEN OTHERS => S_1;

END CASE;
```

Figure 3-2: Equivalent Ada Case Construct

Since the J73 parse tree (Figure 3-3) terminal nodes, when read from left-to-right, do not form a valid string in the target language (Ada), it must be transformed to produce a valid Ada string as shown in Figure 3-4. Those terminal nodes in the J73 parse tree that contain terminals not common to Ada are deleted. Nodes 3, 4, 5, 8, 9, 13, 16, and 18 are deleted. Notice that the node 3 J73 terminal, (;), can be

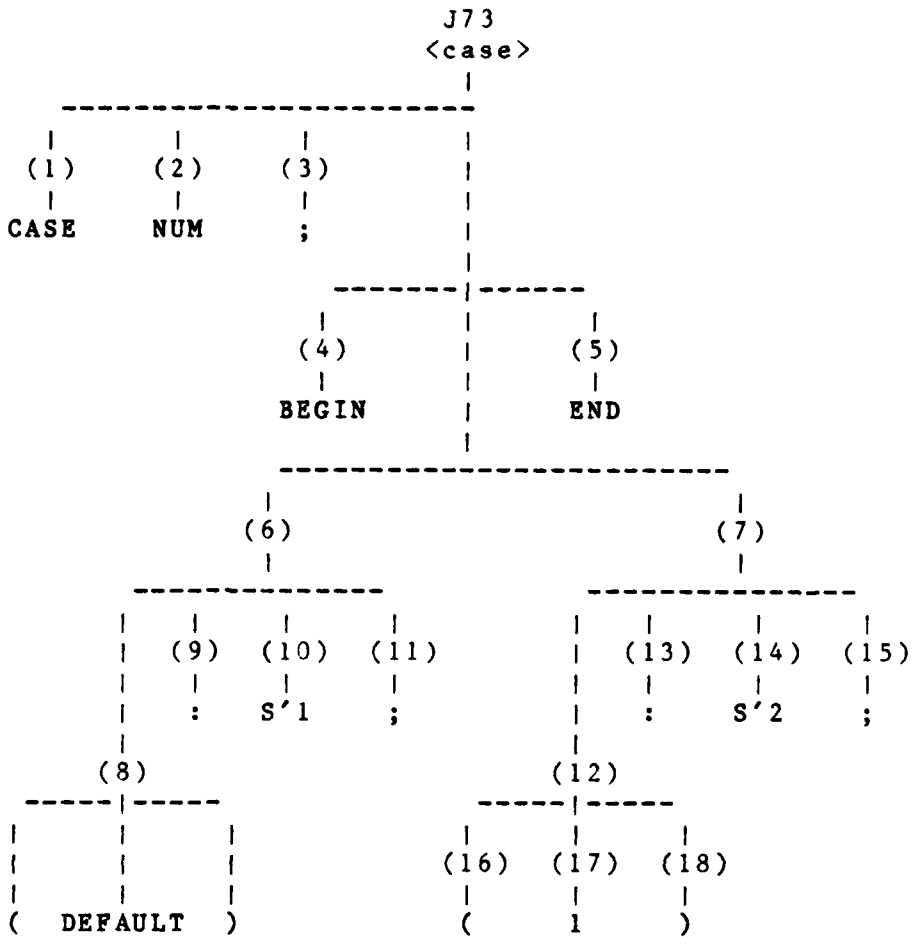


Figure 3-3: J73 Parse Tree for Case Statement

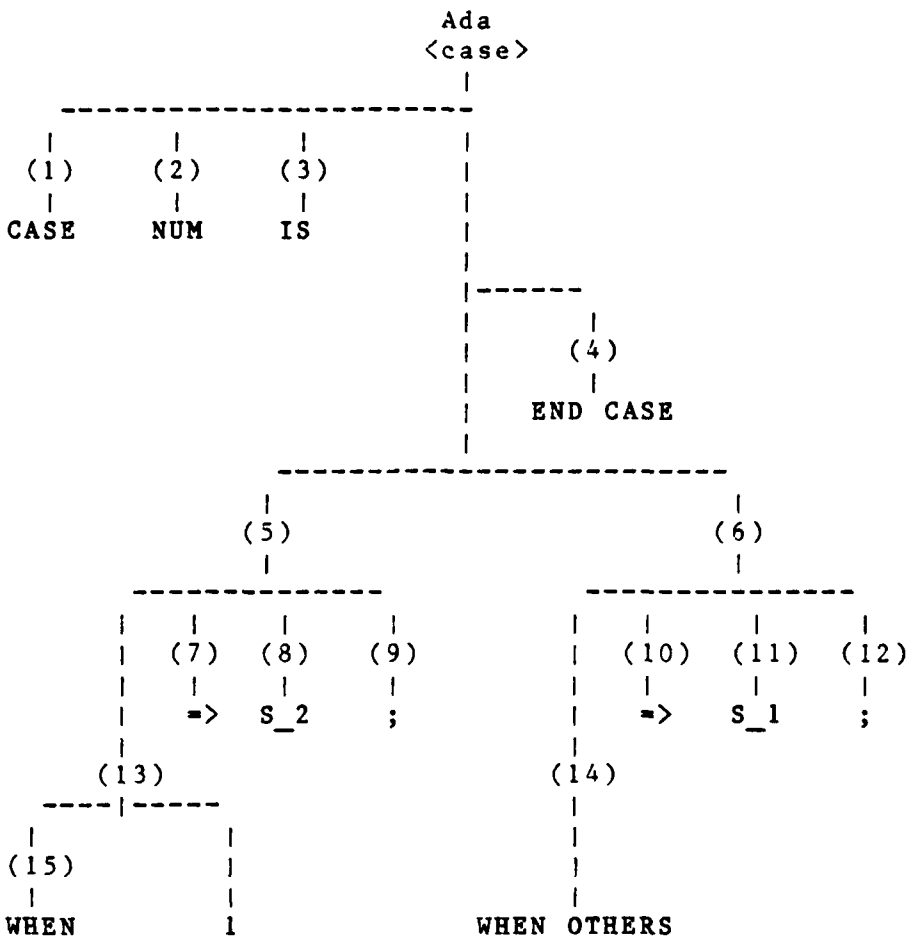


Figure 3-4: Ada Parse Tree for Case Statement

seen as being deleted and the Ada terminal, (IS), being added in its place. The subtrees at interior nodes 6 and 7 are swapped to place the J73 terminal, (DEFAULT), as the last alternative in the case statement since the corresponding Ada terminal, (WHEN OTHERS), must appear as the last case alternative when present. Refer also to Section 3.7. The new nodes that add unique Ada terminals to the Ada parse tree (Figure 3-4) are 3, 4, 7, 14, and 15.

During the analysis of the specific software functions,

several areas were found to be deficient. The first and most obvious deficiency in JATS was that it translates only a subset of the J73 syntax. This was naturally expected due to the scope and time limits imposed on the original developer. There was a lack of in-line documentation contained in the JATS source file. A not so obvious problem was the obsolete input/output (I/O) interface that was embedded throughout JATS. Several subtle deficiencies were also discovered during the analysis. These include recognition and handling of nonprintable characters, upper/lower case sensitivity, inclusion of terminal symbols in the J73 grammar that did not belong, unnecessary numeric literal conversions, and overflow errors during table lookup operations. A serious problem also surfaced in the area of the system stack management. Also an error in the transformational process of creating the equivalent Ada <CASE> construct was noted. It was also seen that JATS received only limited testing. Each of these areas will be discussed in greater detail in the following sections.

3.2 JATS Translates Subset of J73

Had it not been for this short coming, no further development would have been necessary or desirable. Since only a subset of J73 is translatable, it is natural to finish as much of the remaining syntax as possible in this current effort.

A brief summary of the major J73 constructs that the

original JATS translated follows:

1. The main program structure
2. Local procedure structure
3. Case statement
4. Simple <BALANCED-IF-STMT>
5. Simple <BALANCED-WHILE-STMT>
6. Item and type declarations
7. Assignment statement

3.3 Lack of In-line Documentation

When the original JATS source listing was initially examined, it lacked critical in-line documentation. In-line documentation consists of comments included in the source file to give the reader a clearer understanding of the intricacies of the code. Without these comments from the original programmer, the functions of the code often have to be guessed and the guesses are often completely wrong or else not very exact. If the code had contained more documentation, the amount of time required to gain a good understanding of the function of the entire program would have been very small when compared to the amount of time required to gain the same level of understanding when the code lacked these vital comments.

Due to this lack of documentation, an inordinate amount of time was spent digging into the details of the source to ascertain its function. Some of these functions may not be exactly accurate since only the originator knew the exact

function to be performed.

3.4 JATS I/O Interface

The initial area of concentration was to get the outdated translator running in the current environment on the DEC-10. The major change required to get the translator running was to rewrite the I/O interface. Brozovic had chosen an alternate PASCAL-like I/O interface rather than using the FORTRAN based I/O interface which was locally maintained along with the current version of the J73 compiler.

The time came when it was necessary to actually execute JATS in its original condition and see first hand exactly how the pieces fit together. Once current relocatable files for the parser tables (Ref 1:16) and for the translator program were obtained using the current version of the JOVIAL compiler, the initial attempt to load the object modules failed. The problem was discovered to be obsolete (not generated with the current version of the compiler) relocatable files for the alternate PASCAL-like I/O interface used by Brozovic.

To compound matters, no documentation on the old I/O functions existed; however, most of the quesses were fairly accurate. After an extensive search, only an old listing was found that explained the functions of the old I/O interface. The next logical step would have been to simply recompile the I/O source code and obtain current relocatable files. A

search for the machine readable source files ended with only an old listing of the source being found. Since it was very questionable whether this source would even function if entered manually, the decision to rewrite the I/O interface using the local standard was the only viable alternative.

3.5 Nonprintable Characters

It was discovered somewhat by accident that certain non-printable characters were omitted as possible delimiter characters. Some of these characters, especially horizontal tab characters (HT), tend to be found in program text files. These characters caused grave difficulties for JATS since their existence was not taken into account. Since they were not considered in the original JATS and therefore not considered valid delimiters in the J73 syntax, reserved words and user defined identifiers could not be found in the symbol table.

3.6 Sensitivity to Upper/Lower Case Response

There are three instances in the main program that accept responses from the user. The original code accounted for only an upper case "Y/y" for a yes response. Since many terminals have upper and lower case keyboards, the translator system should accept either an upper or lower case "Y" for a yes response.

3.7 <CASE> Translation Error

An error was discovered in the <CASE> statement translation (Ref 1:38). The error was in the ordering of the

<WHEN OTHERS> case alternative in the output Ada source. In J73 (MIL-STD-1589B), the corresponding structure, <DEFAULT>, can appear anywhere in the list of case alternatives; however, Ada specifies that this default alternative, <WHEN OTHERS>, must appear as the last alternative when used (Ref 11:5-5). The syntax of the Ada case statement shown in Appendix F is incorrect due to this ordering error.

3.8 Integer Overflow

Once the original translator was operating for a test J73 input program, a problem of integer overflow occurred in the table lookup procedure. This procedure used a bit pattern representation of the odd numbered characters in a user defined <name> to obtain an integer hash index into the user defined symbol table. Each character's bit pattern was converted to integer. The hashing key was successively multiplied by each character's integer value until all odd characters had been utilized.

The integer overflow did not actually hurt the operation of the translator, but the integer overflow interrupt generated by the operating system (OS) caused the JOVIAL Interactive Debugger (JID) to abort. This debugger was very useful for obtaining execution-time snapshot dumps of the executing translator at user defined breakpoints; however, it could not be used while the overflow problem existed. The cause of the overflow was never determined.

3.9 System Stack Management

A serious problem was encountered when the system maintained stack overflowed. The OS issued a message that indicated that the user push-down stack had exceeded its limits. The compiler maintains the number of words reserved for the push-down stack and the top of stack address (stack pointer) in register 017. Upon examining this register before execution, it showed that only 64 words had been allocated to the stack. This amount was simply insufficient to do any meaningful amount of processing, much less supporting recursive procedure calls where local variables, program counters, and other necessary data must be saved on the stack.

All avenues of approach were tried in an attempt to request the compiler to allocate a larger amount of memory to the stack. According to the sponsor, the only way to get the compiler to allocate a user specified amount of memory to the stack would require a modification to the compiler with a new version being released. This was clearly not a desirable solution. Other compilers had parameters that a user could set which allocate fixed amounts of stack space, but JOVIAL(J73) does not. Once the available user address space had been exhausted for use as stack space, the compiler should request more space in one page increments (512 words). The compiler did not appear to request any additional stack space, but allowed the stack to overflow.

The maximum address space per user on the DEC-10 is 512 pages of memory (approximately 256K words). The initial load of the translator required only 153 pages. When the stack overflow error was issued, the memory size of the translator was only 156 pages, an increase of three pages. This seemed to indicate either an OS or compiler problem which should have been transparent to the user.

Major Caro and Mr. Mike Burlakoff did indicate that an earlier version of the J73 compiler did experience problems with push-down stack management during recursive procedure calls. The problem was supposed to have been corrected earlier, but they checked to be sure and it was properly corrected.

The project sponsor thought that the stack overflow error was caused by a bad algorithm in JATS and suggested the algorithm be examined and modified to require less memory. However, the algorithm worked perfectly for a smaller test program translation in which the level of recursion was 20 deep. The stack overflow was occurring at a recursion depth of only 24. No definitive corrective measures were obtained after several weeks of working with the sponsor and gathering what information was available. A detailed examination of the project workbook for the development of the JOVIAL J73/I compiler revealed little helpful information. The workbook provided only high level information, not the detailed information that was required to correct the stack overflow

problem.

Mr. Bob Engimann, TRW Inc., the primary software contractor for the JOVIAL(J73) compiler and Software Engineering Associates (SEA) in California, the actual compiler designers subcontracted by TRW, was contacted to try and resolve this problem. After approximately one hour of conversation with SEA, a corrective solution was provided.

Much of the information gathered, until the exact reason for the stack overflow problem was determined by a coordinated TRW/SEA analysis, was either erroneous or misleading. The JOVIAL(J73) compiler does maintain a dynamic stack to preserve local variables and temporary compiler generated data, but the stack that is maintained for procedure linkages (return addresses) is a static stack. One word per procedure call is required for this linkage. The JOVIAL program initialization routine from the JOVIAL runtime library called the FORTRAN initialization routine which set the static stack size at 64 words. This meant that only 64 nested procedures could be invoked, including recursive procedure calls, before the stack overflowed.

3.10 Error in Single Character Terminal Symbols

The period (.) was inadvertently included as a J73 terminal symbol. It is not one of the terminals defined in the J73 grammar. Refer to Appendix D for a complete list of terminal symbols for J73.

When a period was encountered during the lexical analysis of the input program text, the parsing algorithm would terminate prematurely. The sequence of events that led to the termination is:

1. The lexical analyzer (input program text scanner) attempted to return the token number for the period. Since a period was not found in the parser vocabulary table, a value of zero was returned.

2. Upon return to the main parsing loop, a reduction was attempted based upon the current (erroneous) token, the period. Therefore, no reduction was possible.

3. Finally, an attempt to find a transition to another state also failed since it was also based upon the current token.

4. Since no transition was possible, the translator terminated with the current state and current token, a <letter> assummed in this case, written into the output list file.

According to MIL-STD-1589B, "<Marks> are used either alone or in conjunction with other characters as operators, delimiters, and separators" (Ref 2:116). A period was not an <operator> (Ref 2:120) and not a <separator> (Ref 2:122). MIL-STD-1589B did not define <delimiter>. The only reference to a period occurred in the definition of <mark> and <numeric-literal> (Ref 2:116,123). As well as could be determined, the period only had significance in the <fractional-form> of a <floating-literal> and <fixed-literal> (Ref 2:123-124). Based upon an examination of the standard, it was assumed that the period was a delimiter but a very special delimiter in the case of the translator. A request

for clarification or correction to MIL-STD-1589B was submitted.

3.11 Erroneous Numeric Literal Conversion

In the original translator <numeric-literals> (integer, floating point, and fixed point) were converted to internal value. When the equivalent Ada source was being output, the integer literals were converted back into ASCII correctly, but the fractional portion of fixed point and floating point literals lost some of its precision for unknown reasons. But, the translator had no need to convert the input numeric literals into internal form because no arithmetic operations were performed on them.

3.12 JATS Received Limited Testing

Brozovic indicated in a telephone conversation that only a small test program had ever been translated by the original JATS. The capacity of the original translator should have been determined and documented in the original thesis. This would have been an obvious area to examine early in the analysis phase.

Since Brozovic used an alternate I/O interface package, the stack might have been dynamic and therefore not caused a push-down stack overflow. Since the current translator utilized the locally standard FORTRAN I/O interface package, this very artificial stack size limit of 64 words was the cause of the problems that appeared to be capacity limits of the original translator. The current translator, however,

was tested for capacity limits and most typical J73 programs should be translatable. See Section 4.10 for the details and limits of the test.

3.13 Summary

Despite these shortcomings of JATS, it was really a very good software product. It performed its task of translating J73 into Ada very elegantly and efficiently. Chapter 4 will discuss the modifications required to correct the deficient areas uncovered during the detailed analysis of JATS, thus making it much more reliable.

4. Continued Software Development

Several areas required modification as pointed out in the analysis of the current translator system in Chapter 3. These software changes actually equate to the completion of the implementation of the algorithm as stated in Chapter 2.

4.1 I/O Interface

The most significant modification was required in rewriting the obsolete I/O interface with the local FORTRAN equivalent. It was no quick and easy task to rewrite the PASCAL-like interface with the locally standard FORTRAN I/O interface. Since the JATS I/O involves primarily character manipulation, FORTRAN does not behave quite as nicely as the PASCAL-like interface. To get around the shortcomings of FORTRAN character manipulation, input and output character buffers were designed and included in JATS. Additionally, software was added to manipulate the input and output buffers. These buffer functions were performed without user intervention with the alternate I/O interface. Initially, integer, fixed point and floating point numbers had to go through a special conversion process to obtain the equivalent ASCII string required for insertion into the output buffer. This newly installed code for number conversions was later abandoned when it was determined to be unnecessary to convert ASCII to internal form and then convert back to ASCII for output. Refer to Section 3.11.

The remainder of the I/O conversion proceeded fairly well but was mostly a trial-and-error process. The documentation describing the standard I/O was very brief and superficial with few examples. The examples were also very misleading due to the way the syntax was typed. Additionally, several procedures had to be added to JATS to emulate the old I/O. The functions of these procedures are described more fully in Appendix C.

4.2 Additional In-line Documentation

In-line documentation was added to the source code when modifications were made. Additionally, prologues were inserted at the beginning of each procedure that described its function .

4.3 Special Nonprintable Characters

Source code was added to take care of special non-printable characters. The horizontal tab, linefeed, and carriage return were added as valid delimiters. Source code was also added to print out the integer value of all invalid characters encountered while processing the input J73 program text file. In this way, an identifying value of all non-printable characters will be output since the character output will print as a blank.

4.4 Upper/Lower Case Sensitivity

The upper/lower case character sensitivity was removed from the interactive responses in JATS.

4.5 <CASE> Translation Error

The correct ordering of the <WHEN OTHERS> case alternative for Ada was not implemented due to time limitations.

4.6 Integer Overflow in Hashing Algorithm

The hashing algorithm was modified to correct the integer overflow problem by adding the binary values of each odd character in a <name> rather than by multiplying the values as was done originally. The data representation (REP) function was not used, but rather an explicit bit conversion was used to obtain the binary value of the characters. No further integer overflow problems have occurred and the hash indexes for several <name>s in a test program were unique although they did not have to be.

4.7 System Stack Modifications

The correction was very simple once TRW/SEA fully understood what was really happening. Mr. Bob Engimann, with direction from SEA, inserted four lines of DEC-10 macro code into the JOVIAL program initialization routine to overwrite the stack size and stack pointer. The size was arbitrarily set at 1000 decimal words. However, the size can be set to any reasonable size required for a particular process as long as the maximum user address space of 512 pages (DEC-10 restraint) is not exceeded. The user need only change the stack size to a new value and reassemble the macro code. This solved the stack overflow problem. Figure A-3 shows the

macro code required for the correction.

4.8 Period (.) Token

The period is no longer considered as a <mark> character. It is considered as an integral part of fixed and floating point numeric literals only. The analysis leading to this conclusion was extensive, but the modification was minor.

4.9 Numeric Literal Conversion

A modification to JATS was made to ensure that the Ada output reflect the exact numeric literals as the input J73 source. The numeric literals are treated as ASCII character strings, stored as ASCII, and finally output in the Ada source program as ASCII. This process replaced the unnecessary conversion processes and allowed approximately 100 lines of source code to be deleted from JATS.

4.10 Current JATS Capacity Testing

JATS was tested with an unrealistically deep level of nested <case> statements. This was done as a simple capacity test of the translator, especially the depth of recursion required during the Ada source program output. Each recursive procedure call uses a word of the static stack for the linkage. The JOVIAL (J73) compiler limits of nested <case> statements were reached long before the translator reached its limits. The compiler aborted after its internal buffers were exceeded. JATS translated this sample J73 program with no problems and reached a recursion depth of

117. The internal buffers in JATS were finally exceeded; however, it did translate a sample program that required a recursive depth of 165. With a 1000 word push-down stack for procedure linkage, JATS should be capable of translating most typical J73 programs.

5. Results and Discussions

The major result of this thesis effort was the development of a framework for future development efforts. There were several problem areas uncovered, some very minor while others were major. Refer to Appendix C for a summary of the major modifications to JATS.

5.1 Framework for Future Development Efforts

The significant portions of this report that establishes the framework for future development efforts are the detailed step-by-step user's/maintenance programmer's guide (Appendix A) and the analysis of the runtime processes (Appendix B). These portions should be very helpful to the JATS maintenance programmer who has already been designated by the sponsor. Essential information is provided in an organized manner and will eliminate the need to guess and the need to dig into the details of the translator to ascertain its functions. The user's guide presents all necessary information needed to recreate any subpart of the translator system and conveys the associations and relationships among the parts.

5.2 System Reliability

After the extensive analysis and subsequent modifications required to complete the correctness and the implementation of the algorithm, the translator system demonstrated a state of increased reliability. Prior to these changes, the software was extremely unreliable (i.e.

prone to frequent software failure during the parsing phase of a typical J73 program).

5.3 Token Values

A small but very important piece of the translator puzzle was the determination of how tokens values are assigned and what each token meant. These token values are referenced throughout JATS but how and by what process they were established was not clearly documented. Since the current grammar was sufficient and would not be changed for now, the parser table generator was not initially run. However, once it was run, one of the outputs was a list of terminals (tokens) and nonterminals with their associated integer values. Appendix D contains the complete list of J73 terminals and nonterminals. The values assigned are vitally important to understanding how the JATS source code functions. The values assigned to the J73 productions, which are also output by the generator, are of equal importance. Brozovic included these in his appendix A (Ref 1:52-71).

5.4 Automatic Translation and Special Features of a Language

As pointed out by Brozovic (Ref 1:37) and others in the literature, automatic translation of one programming language into another is never quite as good as if the program is simply written by hand or translated manually into the target language. The translated program will perform the same functions as the original, but will not take advantage of the special features in the target language. An example would be

the exception handling features of Ada which would not be reflected in any J73-to-Ada translated program.

5.5 I/O Interface

An area that required significant effort and much time was the conversion from the alternate PASCAL-like I/O interface to the locally maintained FORTRAN standard. Several new procedures were added to the translator, as well as, code to manipulate an input and an output buffer. This new I/O interface should cause no further problems since the runtime routines that support it are maintained along with the JOVIAL(J73) compiler.

5.6 Problems Associated with <DEFINE-DECLARATION>s

The <define-declaration> in J73 will cause severe problems for the translator. A **DEFINE** "is used to associate a name with a text string" (Ref 2:42). The translator attempts to parse the user defined name rather than the actual textual string it represents. As a result, the parser terminates prematurely because the user name is not considered valid J73 syntax. A possible solution to this problem could be to preprocess the J73 source programs that contain <define-declaration>s and substitute the actual text in place of the name and delete the declarations from the J73 source. The preprocessed source could then be translated in the normal manner.

5.7 User Defined Stack Size

Mr. Bob Engimann, TRW, was asked if some external mechanism could be added to allow the user to specify the stack size when the default of 64 was not sufficient. He said that they would provide such a mechanism and utilize the same macro code (software patch) as shown in Figure A-3. The IFNDEF macro call was used for this reason. It specified to define the stack length only if it had not been previously defined, for example, by a user request and thus not overwrite the user specified size by the constant decimal 1000 value. The question of a dynamic stack for procedure linkage was raised, but TRW said that the stack would have to remain static for now since the compiler linkage mechanism might have to be modified to support a dynamic stack. The larger static stack worked very well since the returned words in the static stack are reuseable.

5.8 DEC-10 System Constraints

This translator is not directly rehostable on other computer systems other than the AFWAL/AA DEC-10 computer system, even though the new host supports JOVIAL(J73). Other DEC-10 configurations may not support the translator. The translator is nonportable because of the machine dependent runtime library routines. These routines also support the I/O interface used in JATS. The current I/O interface would have to be removed from the translator and a new interface substituted to allow the translator to execute on other computer systems that support JOVIAL(J73). This would be a

very time consuming process since essentially the same process was done for the current translator. Refer to Section 3.4.

5.9 Current JATS Deficiencies

Due to the many deficiencies found and modifications that had to be made in order to improve the reliability of JATS in the current environment, no additional J73 language productions were translated. Additionally, the grammar changes in MIL-STD-1589B were not incorporated.

5.9.1 No New Language Productions Translated

The recommendations presented by Brozovic (Ref 1) are still valid. These recommendations should be included in JATS as soon as possible to provide a more useful product to the Avionics Laboratory. Refer to Brozovic's Appendix B (Ref 1:52) for a list of the J73 productions. Only the original subset of the J73 language is translatable. See Chapter 2 for a summary of the major J73 language constructs in the translatable subset.

5.9.2 MIL-STD-1589B Not Implemented

JATS translates J73 as defined by the outdated MIL-STD-1589A. The current structure of the parsing tables will not allow the new elements of J73 defined in the B version standard to be parsed and therefore, these elements are not translatable. In order for JATS to translate the B version grammar, the J73 grammar defined in BNF will have to be modified to include the new productions. Time did not

permit the B version to be implemented. Refer to Section A.3 for a discussion of regenerating the parser tables when the grammar requires modification.

There are several new reserved words added in the B version which currently have no meaning in J73. These were added to maintain upward compatability with future extensions of J73. Since they are currently meaningless, they should not be included as reserved words in the J73 BNF definition of the grammar (Ref 2:119-120). Only reserved words that actually appear in some production of the grammar need be included. Because of the few changes in the B version, it is recommended that the current parser tables for version A continue to be used. The new B version elements can be translated manually rather than having the maintenance programmer regenerate the parser tables for these minor changes.

However, when the J73 grammar requires a modification, the following version B reserved words and elements should be included in the J73 BNF since the parser tables will have to be regenerated.

1. Implementation Parameters and Size Functions

MAXTABLESIZE
INTPRECISION
BYTEPOS

2. Parameters and Function Names

BYREF
BYVAL
BYRES

3. Define Listing Directives

!LISTINV
!LISTEXP
!LISTBOTH

4. Square Brackets

[ ]

Since the square brackets are implementation dependent, their occurrence in a J73 program being translated could better be handled by JATS software rather than by a grammar change. A solution to this dilemma could be handled by substituting the token value for the symbol, (*, for the input symbol, [. Likewise, the token value for the symbol, *), could be substituted for the input symbol,]. This would, in reality, be fooling the parser, but the parsing would continue as if it had encountered the brackets (* *) used for type conversions (Ref 2:117).

6. Recommendations for Future Study

Since the original translator system is capable of translating only a subset of J73 as defined in MIL-STD-1589A, the bulk of the J73 syntax has not yet been translated. Since JATS will be an ongoing project, there are several areas that should be analyzed and included in later versions of the translator if they are considered desirable and feasible. These areas for consideration are further divided into those essential, those desirable, and those that should not be included. Refer also to Brozovic (Ref 1:41-43).

The areas considered essential are listed below.

1. Complete the Translation Module. Since the framework has hopefully been well established, the completion of the translation part of JATS should proceed smoothly allowing much more of the J73 syntax to be translated without too much difficulty. This capability can be added to the translator without disturbing any existing source code.

2. Incorporate External Files. The primary externally referenced file appears to be the file specified by the !COMPOOL directive. This file can be treated as any other J73 program source file since, in reality, its exactly that. The !COMPOOL file would possibly involve adding the symbols to the user symbol table initially so as to allow those names to be resolvable once their reference was encountered in the J73 source program. The other often used external referenced file is the file specified by the !COPY directive. This directive is used to include J73 source files at the point of invocation by the !COPY directive. This directive could be handled by simply opening the external file, suspend reading from the original source file, and initiate reading from the newly specified external file. Once the external file is exhausted, reinitiate the reading of the original.

3. Retain Inline Documentation. Since inline documentation in the form of comments is considered very important to any source text, provisions should be made to retain this vital information. Once again the importance of this documentation cannot be overstated. This enhancement would require a larger area to be established to store these ASCII character strings. A comment node could be created as the last node of a source line that contained a comment.

An area that is considered desirable but one that should not be implemented immediately is the inclusion of the grammar changes to the J73 syntax contained in MIL-STD-1589B. It is felt that these changes should not be made until they become necessary or until the grammar requires modification for other reasons. See Section 5.9.2 for a further discussion. The changes in the B version are considered minor and would affect the translation process only minimally.

An area considered nonessential and undesirable was making JATS interactive as proposed by Brozovic. He proposed making the translator "interactive and allowing the user to identify sections of code that require manual translation" (Ref 1:42). This would be one of the nice to have features, but it is felt that its utility would not justify the effort required for its implementation. This would encourage the user to "program at the terminal"; a practice that should always be avoided.

Bibliography

1. Brozovic, Richard L. "JOVIAL(J73) TO ADA TRANSLATOR SYSTEM." Unpublished MS thesis. School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio, December 1980.

2. MIL-STD-1589B. MILITARY STANDARD JOVIAL (J73). Washington, D. C. : Department of the Air Force, June 6, 1980.

3. Defense Advanced Research Projects Agency. REQUIREMENTS FOR Ada PROGRAMMING SUPPORT ENVIRONMENTS : STONEMAN. Arlington, Va. : Department of Defense, February 1980.

4. Glass, Robert L. "FROM PASCAL TO PEBBLEMAN ... AND BEYOND," Datamation, 25 : 146-150. (July 1979).

5. Barrett, William A. and John D. Couch. COMPILER CONSTRUCTION: THEORY AND PRACTICE. No city of publication : Science Research Associates, 1979.

6. Donovan, John L. and Henry F. Ledgard. "A formal system for the specification of the syntax and translation of computer languages," Proceeding Fall Joint Computer Conference, AFIPS Conference Proceedings, 31. 553-569. New York : American Federation of Information Processing, November 1967.

7. Ginsberg, Seymour and Erica M. Rounds. "Dynamic Syntax Specification Using Grammar Forms," IEEE Transactions on Software Engineering, 4, No 1 : 44-55. New York : The Institute of Electrical and Electronics Engineers, Inc., January 1978.

8. Kernigan, Brian W. and P. J. Plauger. Software Tools. Reading, Massachusetts : Addison-Wesley Publishing Company, 1976.

9. Jonas, Ronald W. "Generalized translation of programming languages," Proceeding Fall Joint Computer Conference, AFIPS Conference Proceedings, 31. 570-580. New York : American Federation of Information Processing, November 1967.

10. Sassaman, William A. "A COMPUTER PROGRAM TO TRANSLATE MACHINE LANGUAGE INTO FORTRAN," Proceeding Spring Joint Computer Conference, AFIPS Conference Proceedings, 28. 235-239. New York : American Federation of Information Processing, Spring 1966.

11. Defense Advanced Research Projects Agency. REFERENCE MANUAL FOR THE ADA PROGRAMMING LANGUAGE. Washington, D.C. : Department of Defense, July 1980.

A. Appendix: JATS User's/Maintenance Programmer's Guide

This appendix is divided into two distinct parts: a common user's guide and a maintenance programmer's guide. It describes how to use the translator and provides the maintenance programmer with information necessary to make modifications to JATS, including recreation of the parser tables when the J73 grammar requires modification.

A.1 How To Use JATS

The translator is simple to use, but there are some things the user must do before attempting to translate a J73 program. They include the following:

1. J73 user defined <name>s cannot be Ada reserved words. The user must ensure that the <name>s do not conflict.
2. The use of the single quote (prime) in J73 <name>s must conform to the rules for use of the underscore in Ada <name>s.
3. The $ allowed in J73 <name>s must be removed or changed since there is no equivalent symbol in Ada (Ref 1:85).

To execute JATS, enter the following run command:

```
.RUN DSKF:JATS [6664,202]

WHAT FILE DO YOU WISH TO TRANSLATE?
* <filename>[.<extension>]

INPUT FILE = LFN.J73  OUTPUT FILE = LFN.ADA

THE MAXIMUM DEPTH OF RECURSION WAS nnnnnn
STOP
```

Next JATS will query the user for the input filename containing the J73 program to be translated followed by an * prompt. The user should respond by entering a valid J73 filename, six characters or less. No extension is required but .J73 should be used. The corresponding output file will be created with the same name as the input filename, except with an extension of .ADA. LFN refers to any local filename.

Next the input program will be translated. Prior to program termination, the maximum depth of recursion will be output as a simple method of determining the approximate amount of push-down stack space that was taken for the recursive procedure linkage for calls by the Ada pretty print procedure.

The translator creates one additional file, JATS.LST, that will contain the input program and informational and error messages. The file with the .ADA extension will contain the equivalent Ada source program from the translated J73 source. Should any portion of the input source code not be translated, brackets will be placed around the untranslated portions. See Appendix F for an example of the bracketed portions of a J73 test program that require manual translation.

JATS' internal buffers were designed to allow large J73 programs to be translated. Refer to Section B.1 for buffer sizes. However, should any of the buffers overflow, notify the maintenance programmer and provide the error message that

was output on the terminal. Refer to JATS System Errors in Figure A-1 for the cause and corrective action (Ref 1:84). Additionally, provide a printout of JATS.LST.

Error Number	Cause/Required Changes
1	Parsing Stack Overflow Increase value of MAX'STACK
2	SYM'STORE'TAB Overflow Increase value of MAX'SYM'STORE
3	HEAP Overflow Increase value of MAX'HEAP
4	TREE Overflow Increase value of MAX'NODES
5	STRING Overflow Increase value of MAX'STRINGS
6	STRING'STORE Overflow Increase value of MAX'STRING'STORE

Figure A-1: JATS System Errors

A.2 JATS Maintenance Programmer's Guide

The maintenance programmer may request additional debug information if so desired. However, item ADDITIONAL'DEBUG'INFO must be set to TRUE and the translator program recompiled. This boolean switch should remain turned off for common user versions and only be turned on for the maintenance programmer's development versions, since this information will be meaningless except for someone very knowledgeable with the translator source code. Once the translator has finished, it queries the user again for responses to obtain additional information. **Note:** Any

response except yes, Y or y, will be taken to be no. Additionally, these requests should be used only when a very small test program is being translated since a large amount of printed data is generated.

```
DO YOU WANT THE PARSE TREE OUTPUT? Y/N
*Y/N

NO YOU WANT THE SYMBOL TABLE OUTPUT? Y/N
*Y/N

DO YOU WANT THE TRANSLATED TREE AND SYMBOL TABLE? Y/N
*Y/N
```

Relocatable Files

The translator system requires several relocatable files. These include JATSXX, JINIT, J73TAB, and JOVLIB. The compool files for JOVIO and J73TAB are also required. The creation of these files are described more fully in the following subsections.

Parser Tables

The parser tables contained in J73TAB.J73 must be compiled using the current version of the JOVIAL(J73) compiler. These large tables are included in JATS through the use of the compool directive. Figure A-2 shows the control file contents necessary to create the relocatable and compool files, J73TAB.REL and J73TAB.CMP, respectively. To execute the control file use the command: DO J73TAB.CTL.

Program Initialization

```
RUN NEW:JOVIAL
J73TAB.REL,J73TAB.LST = J73TAB.J73,J73TAB.CMP
/ACROSS/NOI/D
```

Figure A-2: Control File to Compile Parser Tables

The JOVIAL program initialization routine is a special version, JINIT.SPL, that has the push-down stack size set at a constant 1000 decimal words rather than the default value of 64 words. Refer to Section 3.9 and Figure A-3 if the push-down stack size (length) must be changed. If the stack size is modified, simply reassemble JINIT.SPL to obtain the new relocatable file. Additionally, the translator system must be reloaded to bring in the new version of JINIT. See the load command in Figure A-4.

Translator Program

The translator program, written in J73, must also be compiled with the current version of the JOVIAL(J73) compiler. This relocatable file is the last portion needed to create JATS. Figure A-4 shows the control file to compile the translator, load it, create the executable module, and set the protection appropriately to allow user access. A common input source filename, JATSXX.J73 is used to allow the control file to be used without modification. The maintenance programmer simply copies the current version of the translator source into the common filename JATSXX.J73 and then executes the control file with the command: DO

```
        .
        .
        .
        JSP     16,RESET        ;call FORTRAN          *
                                ;initialization        *
         0                                             *
        MOVE    17,PDL          ;increase PDL stack
                                ;load reg 017 with IOWD
                                ;at label PDP
        .
        .
        .
        IFNDEF  PDLEN,          ;if PDLEN not defined
        PDLEN   =^D1000         ;then define as 1000
PDL:    BLOCK   PDLEN           ;declare PDL as block
                                ;with length = PDLEN
PDP:    IOWD    PDLEN,PDL       ;format IO word
                                ;for register 017.
                                ;[neg length and address]
        .
        .
        .
```

* Indicates part of original code

Figure A-3: Macro Code Correction for PDL Overflow

JATSXX.CTL. The files JOVLIB and JOVIO are provided by the sponsor.

```
RUN NEW:JOVIAL
JATSXX.REL,JATSXX.LST = JATSXX.J73/ACROSS/NOI/D
^C^C

LOAD /REL JINIT,JATSXX,J73TAB,/SEARCH NEW:JOVLIB
SAVE JATS
^C^C

PROTECT JATS.EXE<255>
^C^C
```

Figure A-4: Control File to Create the JATS System

A.3 Regeneration of Parser Tables

When a modification to the J73 grammar is required, the parser tables, J73TAB.J73, must be recreated to reflect the grammar change. This will be quite a lengthy process and will involve a lot of work. This section is provided to hopefully guide the JATS maintenance programmer through the details of regenerating the parser tables. Brozovic provides additional information on table generation (Ref 1:12-16).

Modifications to the BNF Grammar Definitions

The first step and perhaps the most difficult is to make the grammar changes required in the BNF definitions of J73 and Ada, files J73BNF.DAT and ADABNF.DAT respectively. The original BNF definitions for the J73 and Ada grammars were provided by Intermetrics, Inc. and Softech, Inc. (Ref 1:13). These definitions define productions in the parsing process of J73 and may require some knowledge of compiling theory to fully understand. In the event that these definitions cannot be successfully modified by the maintenance programmer, Intermetrics or Softech should be contacted for assistance. Once the modified grammar has been obtained, it must be processed by the LR : Automatic Parser Table Generator (Ref 1:46-51).

Using the LR: Automatic Parser Table Generator

This program is a FORTRAN program, LRS.FOR, that accepts a context free grammar in BNF and produces parser tables.

See Brozovic's Appendix A (Ref 1) for a more detailed description of this table generator and the interrelationships among the several parser tables. The two BNF grammar files, J73BNF.DAT and ADABNF.DAT are the required input files. To execute the parser table generator use the following command: DO LRS.CTL. Figure A-5 shows the control file necessary for the compilation, load, subsequent execution, printing of generated listing, and deletion of relocatable files and the executable module.

```
LOAD LRS,ICHR
SAVE LRS
DELETE LRS.REL,ICHR.REL
RUN LRS
PRINT J73LST.DAT/DELETE
DELETE LRS.EXE
```

Figure A-5: Control File to Create Parser Tables

The output file that contains the parser table data is J73TAB.DAT. This data file contains raw data and requires further processing to obtain J73 tables that are compilable. TABLES.J73 is the program that creates the J73 tables from the raw data.

Creation of the Constant J73 Tables

TABLES.J73 creates the constant J73 tables from the raw data contained in the file, J73TAB.DAT. However, TABLES.J73 will not execute in its current condition. It too has the alternate I/O interface embedded throughout. The I/O must be

converted to JOVIO, the locally maintained standard. Once the interface has been converted, run TABLES.J73 and the required constant J73 tables will be output into the file, J73TAB.J73. Now these tables may be compiled as shown in Figure A-2.

A.4 Backup Tapes

The importance of maintaining good configuration control cannot be overstated. One phase involves preserving the contents of the development source files periodically. This was accomplished three times during this effort. All files in the directory DSKF [6664,202] were saved (backed up) on magnetic tape. These tapes are stored in the DEC-10 computer room at the Avionics Laboratory. A listing of the files contained on each of the backup tapes will be given to the sponsor when this project terminates.

The first backup tape was created on July 24, 1981. The identifying information is: reel number = 406, title = EAST, and protection = <055>.

The second backup tape was created on October 3, 1981. The identifying information is: reel number = 481, title = JATS03, and protection = <255>.

The third and final backup tape was created on December 2, 1981. The identifying information is: reel number = 520, title = JATS25, and protection = <255>.

B. Appendix: Analysis of JATS Runtime Processes

The purpose of this appendix is to provide some insight into the portions of JATS that must be immediately available or reside in main memory during different stages of the translation process. This information will be useful in later stages of the continued development of the translator system as larger and larger J73 programs are being translated. It will also be helpful should the translator system be hosted on a smaller computer system. At some point the JATS maintenance programmer may need to reuse portions of the address space once the code and data in those areas are no longer needed. Dynamic use of the user address space is very appealing thus allowing the size of the translator elements to increase and decrease as necessary while the translator is executing.

According to the sponsor, however, dynamic program segmentation is only possible at the assembler language level and not available to JOVIAL(J73) at the high order programming language level. If dynamic program segmentation was available for J73, the user could segment the translator and cause only the required portions to reside in main memory at any given time.

The term "stage" will be used to describe an instant of time that corresponds to the time when certain portions of the translator must be immediately available or reside in

main memory. The number that follows some of the program elements (static data structures in this case) represents its relative size in decimal words. Only fixed size tables will have their associated sizes appearing after their names. The actual sizes of the procedures that comprise JATS are not provided.

B.1 Stage 1

This stage corresponds to the initial execution of the translator in which the input J73 program is being parsed and the parse tree is being created. Figure B-1 shows the stage 1 program flow. The following procedures and tables must be immediately available for execution or reference.

1. All Parser Tables-J73TAB.J73: ENT--1202, LEN--686, LHS--686, LS--2941, LSET--129, NSET--708, FRED--1203, TRAN--22122, VOCAB--224, VSTRING--1305, FTRN--1203, and PROD--708. The total size of these tables currently is 33120 decimal words. The size of VSTRING could be significantly reduced (261 words) by specifying a tight structure where five characters would be packed into each word rather than each character occupying an entire word (Ref 2:27).

2. JATS Internal Buffers (dimensioned tables): STACK--300, NESTING--26, TREE--2500, NODE'TRAN--2500, FREE'TAB--0, HEAD'TAB--128, SYM'STORE'TAB--10000, STRING'TAB--500, STRING'STORE'TAB--10000, and HEAP--10000. The two tables, SYM'STORE'TAB--2000 and STRING'STORE'TAB--2000, could be significantly reduced in size by a factor of five by specifying a tight structure for the tables. The new sizes have been indicated following their names. FREE'TAB's size has been shown as zero since it has the same starting address as does TREE, via an OVERLAY statement.

3. The main program containing the parsing loop and the following procedures: DEC'HEAD, DO'REDUCT, DO'TRAN, ERROR, FIND'REDUCT, FIND'TRAN,

INITIALIZE, LOOKUP, NEWP, SCAN, GET'CHAR, NAME'RESWORD, CHAR'STRING, COMMENT, DIRECTIVE, DOUBLE'MARK, FINDTOK, DIGIT, MARK, and SEMANTIC.

Note: Procedure INITIALIZE is only required during the early phases of Stage 1 to set up the I/O files and get the input J73 filename. Currently table NESTING is not used by the translator. ENDLINE and CLOSE'FILES are used by several procedures when error conditions occur and must always be available.

B.2 Stage 2

This stage is entered once the main parsing loop has terminated. The procedures used in stage 1 are no longer required. The majority of the parser tables (31591 words) are no longer required. The maintenance programmer may request additional information as pointed out in Section A.2. The J73 parse tree will be transformed into the equivalent Ada at the end of this stage. Figure B-2 shows the Stage 2 program flow. The following procedures and tables must be immediately available for execution or reference.

1. Parser Tables: Only VOCAB and VSTRING are required. Their total size is 1529 decimal words.
2. JATS Internal Buffers: All as shown in stage 1 except STACK.
3. The main program and the following procedures: PRINT'SYM'TAB, PRINT'STRINGS, PRINT'TREE, PRINT'NODE, TRAN'TREE, TRANSLATE, ADD'NODE, CHANGE'NODE, DELETE'NODE, DELETE'SUBTREE, MOVE'PTR, and SWAP'NODES.

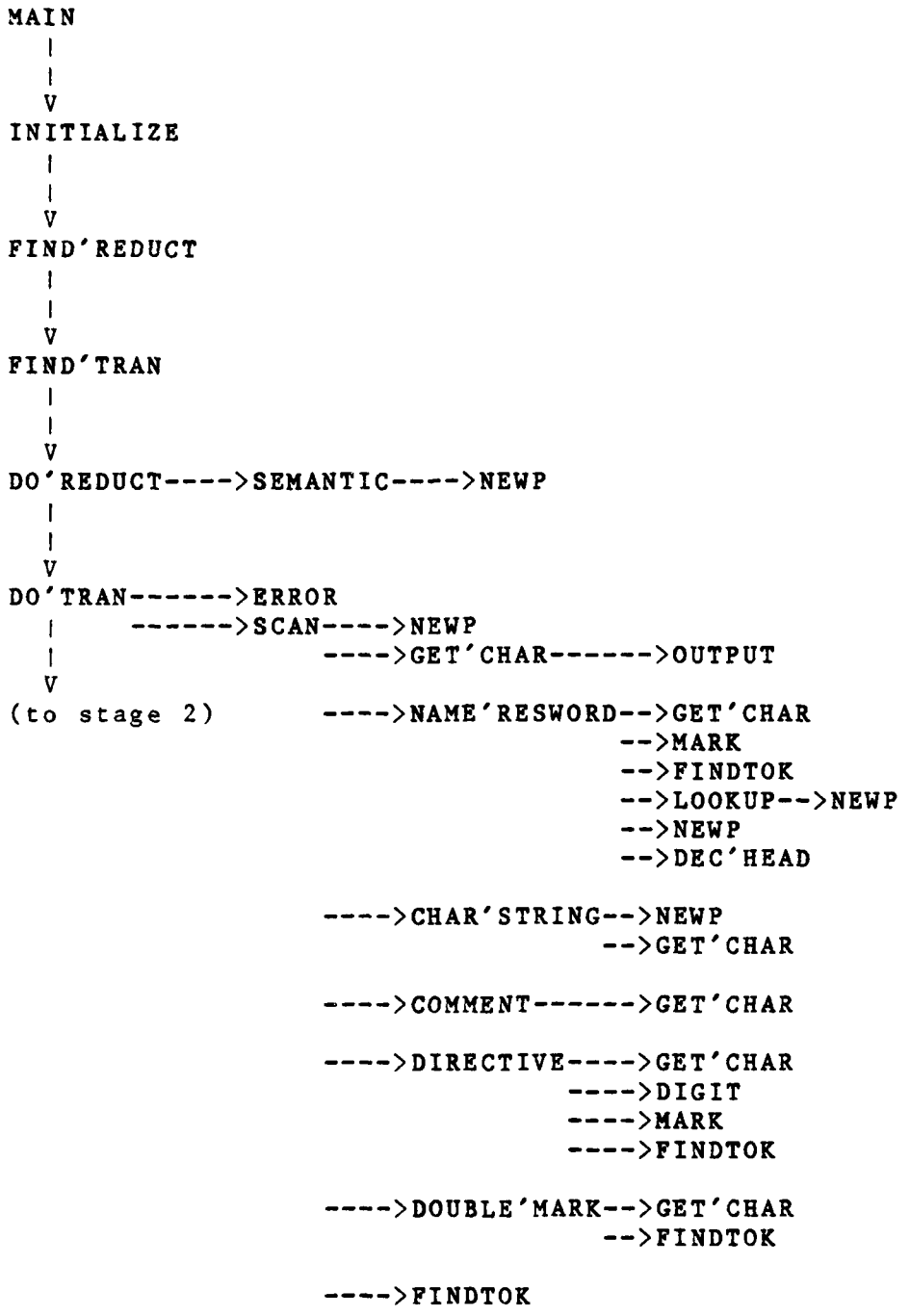


Figure B-1: Stage 1 Program Flow

Figure B-2: Stage 2 Program Flow

B.3 Stage 3

This is the final stage of the translator's execution. The input program has been translated and now the equivalent Ada source text remains to be output. Figure B-3 shows the Stage 3 program flow. The following procedures and tables must be immediately available for execution or reference.

1. Parser Tables: Same tables as in stage 2.
2. JATS Internal Buffers: All except STACK and HEAD'TAB.
3. The main program and the following procedures: PRINT'ADA, PRETTY'PRINT, OUTPUT, ENDLINE, ENDPAGE, TABTO, and CLOSE'FILES.

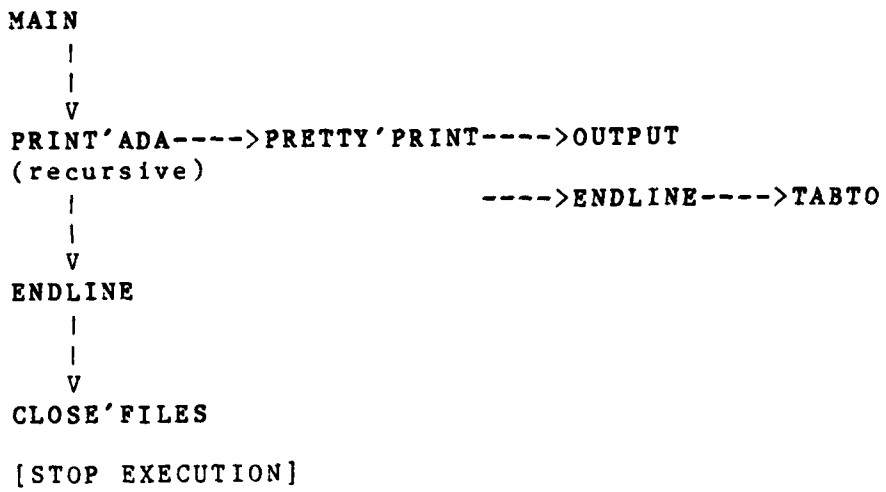


Figure B-3: Stage 3 Program Flow

C. Appendix: Summary of Major Modifications

A brief summary of the major modifications to the original JATS and a short description of the new procedures are provided. See Brozovic (Ref 1:78-83) for descriptions of the procedures that comprised the original JATS. Procedure names appear in bold face type.

The major modifications by procedure name follow:

1. Entire Program. Converted the alternate PASCAL-like I/O interface to the locally standard FORTRAN I/O. Refer also to Section 3.4 for a further discussion. Five new procedures were added to emulate the PASCAL-like I/O and are described toward the end of this appendix. The following procedure calls are to the FORTRAN I/O runtime routines via DEFINEs in the JOVIO compool: WRITE, READ, OPEN, CLOSE, TTY, ACCEPT, ENCODE, AT'END, OUTI, OUTF, OUTC, INC, BLK, and END'L.

2. MAIN PROGRAM. Modified to accept upper/lower case Y for a yes response.

3. INITIALIZE. Modified to get the input file name and create the corresponding output filename with .ADA extension. Format for the file OPEN command had to be changed.

4. LOOKUP. The hashing algorithm that returned indexes into the user symbol table was changed to correct an unknown integer overflow problem. Refer to Section 3.8. The integer values of the odd numbered characters in a <name> were successively added rather than multiplied as in the original algorithm.

5. PRETTY'PRINT. The meaning of link type of 1 was changed to those internally generated integer values during the translate (PROC TRANSLATE) process of JATS. This procedure was further modified to ignore floating point or fixed point literal conversion to internal value. All <numeric-literal>s are treated as ASCII strings without any conversion required. Refer to Section 3.11.

6. **SCAN.** The integer (ASCII) value of invalid characters including special nonprintable characters are output. Horizontal tab (HT), carriage return (CR), and linefeed (LF) characters were added as valid delimiters rather than them being output as invalid. The nontoken character, period, was removed from the case alternative testing for single character tokens. It was added to the case alternative for <numeric-literal>s (ie. floating/fixed point fractional portion).

7. **CHAR'STRING.** Modified to treat <numeric-literal>s as character strings and store them in their ASCII representation. Sets the correct token values for integer or real literals in item **SCAN'TOK.**

8. **GET'CHAR.** Modified to manipulate an 80 character input buffer.

9. **MARK'CHAR.** Nontoken character, period, was removed from the case alternative since it was neither a separator nor an operator. Added HT, CR, and LF as valid delimiters.

10. **NUMBER.** Deleted. Previously performed <numeric-literal> conversions.

11. **CHAR'VAL.** Deleted. Previously used by **PROC NUMBER** during <numeric-literal> conversions.

These new procedures were added to JATS to emulate the old PASCAL-like I/O interface. The functions of these procedures are described below:

1. **OUTPUT.** This global procedure controls the output print buffer. Characters are placed into the buffer one by one. Once the buffer is full or an end of line (EOL) condition is detected, the buffer is written to the specified output file.

2. **ENDLINE.** This global procedure causes the current output print buffer, full or not, to be written. It simply sets EOL to TRUE and writes to the buffer. The new beginning tab location is set for the next output character. The tab position is used while **PROC PRETTY'PRINT** is outputting the equivalent Ada source text.

3. **ENDPAGE.** This global procedure outputs a form feed character to the current output file (unit).

4. **TABTO.** This global procedure moves the index position to **TAB'MARK** in the output print buffer. Used primarily by **PROC PRETTY'PRINT** for proper indentation of the output Ada source.

5. **CLOSE'FILES.** This global procedure outputs the maximum depth of recursion message and closes all input and output files used by JATS.

D. Appendix: List of J73 Terminals and Nonterminals

The integer to the left of the terminals and nonterminals are assigned by the LR parser table generator. These values are referenced through JATS to refer to specific elements of the J73 grammar. A complete list of terminals or tokens and non terminals for the J73 grammar follows:

TERMINALS		NON TERMINALS	
1	!BASE	166	<ABORT-STMT>
2	!BEGIN	167	<ABS-FUNCTION>
3	!COMPOOL	168	<ABSOLUTE-ADDRESS>
4	!COPY	169	<ALLOCATION-SPECIFIER>
5	!DROP	170	<AND-FORMULA>
6	!EJECT	171	<ASSIGNMENT-STMT>
7	!END	172	<BALANCED-FOR-STMT>
8	!INITIALIZE	173	<BALANCED-IF-STMT>
9	!INTERFERENCE	174	<BALANCED-STMT>
10	!ISBASE	175	<BALANCED-WHILE-STMT>
11	!LEFTRIGHT	176	<BALANCED>
12	!LINKAGE	177	<BASE-DIRECTIVE>
13	!LIST	178	<BEGIN-DIRECTIVE>
14	!NOLIST	179	<BIT-FORMULA>
15	!ORDER	180	<BIT-FUNCTION>
16	!REARRANGE	181	<BIT-ITEM-DESCRIPTION>
17	!REDUCIBLE	182	<BIT-LITERAL>
18	!SAFE	183	<BIT-PRIMARY>
19	!SKIP	184	<BLOCK-BODY-OPTIONS-LIST>
20	!TRACE	185	<BLOCK-BODY-OPTIONS>
21	(	186	<BLOCK-BODY-PART>
22	(*	187	<BLOCK-DEC>
23	)	188	<BLOCK-PRESET>
24	*	189	<BLOCK-TYPE-DEC>
25	*)	190	<BOOLEAN-LITERAL>
26	**	191	<BOUNDS-FUNCTION>
27	+	192	<BRANCH-FALSE>
28	,	193	<BY-PHRASE>
29	-	194	<BYTE-FUNCTION>
30	/	195	<CALL-PREFIX>
31	:	196	<CASE-ALT>
32	;	197	<CASE-BODY>
33	<	198	<CASE-CHOICE>
34	<=	199	<CASE-CLAUSE>
35	<>	200	<CASE-INDEX-GROUP>
36	<BLOCK-PRESET-LIST>	201	<CASE-INDEX>
37	<BLOCK-TYPE-NAME>	202	<CASE-STMT>

38 <CHARACTER-LITERAL>
39 <CHARACTER-STRING>
40 <INTEGER-LITERAL>
41 <ITEM-TYPE-NAME>
42 <LETTER>
43 <NAME>
44 <PROC-LABEL-NAME>
45 <REAL-LITERAL>
46 <SYMBOL-LIST>
47 <TABLE-TYPE-NAME>
48 =
49 >
50 >=
51 @
52 A
53 ABORT
54 ABS
55 AND
56 B
57 BEGIN
58 BIT
59 BITSINBYTE
60 BITSINPOINTER
61 BITSINWORD
62 BITSIZE
63 BLOCK
64 BY
65 BYTE
66 BYTESINWORD
67 BYTESIZE
68 C
69 CASE
70 COMPOOL
71 CONSTANT
72 D
73 DEF
74 DEFAULT
75 DEFINE
76 ELSE
77 END
78 EQV
79 EXIT
80 F
81 FALLTHRU
82 FALSE
83 FIRST
84 FIXEDPRECISION
85 FLOATPRECISION
86 FLOATRADIX
87 FLOATRELPRECISION
88 FLOATUNDERFLOW
89 FOR
90 GOTO
91 IF

203 <CHARACTER-ITEM-DESCRIPTION>
204 <COMPLETE-PROGRAM>
205 <COMPOOL-DEC-LIST>
206 <COMPOOL-DEC>
207 <COMPOOL-DECLARED-NAME-LIST>
208 <COMPOOL-DECLARED-NAME>
209 <COMPOOL-DIRECTIVE-LIST>
210 <COMPOOL-DIRECTIVE>
211 <COMPOOL-FILE-NAME>
212 <COMPOOL-MODULE>
213 <COMPOUND-BODY>
214 <COMPOUND-COMPOOL>
215 <COMPOUND-DEF>
216 <COMPOUND-END>
217 <COMPOUND-PRESET-SUBLIST-HEAD>
218 <COMPOUND-PRESET-SUBLIST>
219 <COMPOUND-PROGRAM>
220 <COMPOUND-REF>
221 <COMPOUND-TERM>
222 <CONSTANT-DEC>
223 <CONVERSION>
224 <COPY-DIRECTIVE>
225 <DATA-DEC>
226 <DEC-LIST>
227 <DEC>
228 <DEF-BLOCK-INSTANTIATION>
229 <DEF-PART>
230 <DEF-SPEC-CHOICE-LIST>
231 <DEF-SPEC-CHOICE>
232 <DEF-SPEC>
233 <DEFAULT-PRESET-SUBLIST-HEAD>
234 <DEFAULT-PRESET-SUBLIST>
235 <DEFINE-DEC>
236 <DIMENSION-LIST-HEAD>
237 <DIMENSION-LIST>
238 <DIMENSION>
239 <DIRECT-COMPOUND-END>
240 <DIRECTIVE>
241 <DROP-DIRECTIVE>
242 <EJECT-DIRECTIVE>
243 <END-DIRECTIVE>
244 <ENTRY-SPECIFIER>
245 <EQV-FORMULA>
246 <EXIT-STMT>
247 <EXPRESSION>
248 <EXTERNAL-DEC>
249 <FACTOR>
250 <FIXED-ITEM-DESCRIPTION>
251 <FIXED-MACHINE-PARAMETER>
252 <FLOATING-ITEM-DESCRIPTION>
253 <FLOATING-MACHINE-PARAMETER>
254 <FOR-BY>
255 <FOR-CLAUSE>
256 <FOR-THEN>

```
 92 IMPLFIXEDPRECISION
 93 IMPLFLOATPRECISION
 94 IMPLINTSIZE
 95 INLINE
 96 INSTANCE
 97 ITEM
 98 LABEL
 99 LAST
100 LBOUND
101 LIKE
102 LISTBOTH
103 LISTEXP
104 LISTINV
105 LOC
106 LOCSINWORD
107 M
108 MAXBITS
109 MAXBYTES
110 MAXFIXED
111 MAXFIXEDPRECISION
112 MAXFLOAT
113 MAXFLOATPRECISION
114 MAXINT
115 MAXINTSIZE
116 MAXSIGDIGITS
117 MAXSTOP
118 MINFIXED
119 MINFLOAT
120 MINFRACTION
121 MININT
122 MINRELPRECISION
123 MINSCALE
124 MINSIZE
125 MINSTOP
126 MOD
127 N
128 NEXT
129 NOT
130 NULL
131 NWDSEN
132 OR
133 OVERLAY
134 P
135 PARALLEL
136 POS
137 PROC
138 PROGRAM
139 R
140 REC
141 REF
142 RENT
143 REP
144 RETURN

257 <FORMAL-DEFINE-PARAMETER-
                    LIST-HEAD>
258 <FORMAL-DEFINE-PARAMETER-LIST>
259 <FORMAL-IO-PARAMETER-LIST>
260 <FORMAL-PARAMETER-LIST>
261 <FORMULA>
262 <FUNCTION-CALL>
263 <FUNCTION-DEC>
264 <FUNCTION-DEF>
265 <FUNCTION-HEADING>
266 <GOTO-STMT>
267 <IF-CLAUSE>
268 <IF-PREFIX>
269 <INITIAL>
270 <INITIALIZE-DIRECTIVE>
271 <INLINE-DEC-HEAD>
272 <INLINE-DEC>
273 <INPUT-LIST>
274 <INPUT-PARM>
275 <INTEGER-ITEM-DESCRIPTION>
276 <INTEGER-MACHINE-PARAMETER>
277 <INTERFERENCE-CONTROL>
278 <INTERFERENCE-DIRECTIVE>
279 <INTRINSIC-FUNCTION-CALL>
280 <INVOCATION>
281 <ISBASE-DIRECTIVE>
282 <ITEM-DEC>
283 <ITEM-PRESET-OPTION>
284 <ITEM-TYPE-DEC>
285 <ITEM-TYPE-DESCRIPTION>
286 <LABEL>
287 <LEFTRIGHT-DIRECTIVE>
288 <LHS>
289 <LIKE-OPTION>
290 <LINKAGE-DIRECTIVE>
291 <LIST-DIRECTIVE>
292 <LIST-OPTION>
293 <LOC-FUNCTION>
294 <LOCATION-SPECIFIER>
295 <LOWER-BOUND>
296 <LTR>
297 <MAIN-PROGRAM-MODULE>
298 <MODULE>
299 <NAME-LIST>
300 <NAMED-VARIABLE>
301 <NEXT-FUNCTION>
302 <NOLIST-DIRECTIVE>
303 <NON-NESTED-SUB-LIST>
304 <NON-NESTED-SUB>
305 <NULL-PRESET>
306 <NULL-STMT>
307 <NWDSEN-FUNCTION>
308 <OR-FORMULA>
309 <ORDER-DIRECTIVE>
```

```
145 S
146 SGN
147 SHIFTL
148 SHIFTR
149 START
150 STATIC
151 STATUS
152 STOP
153 T
154 TABLE
155 TERM
156 THEN
157 TRUE
158 TYPE
159 U
160 UBOUND
161 V
162 W
163 WHILE
164 WORDSIZE
165 XOR
```

```
310 <ORDINARY-ENTRY-SPECIFIER>
311 <ORDINARY-TABLE-BODY>
312 <ORDINARY-TABLE-ITEM-DEC>
313 <ORDINARY-TABLE-OPTIONS-LIST>
314 <ORDINARY-TABLE-OPTIONS>
315 <OUTPUT-LIST>
316 <OUTPUT-PARM>
317 <OVERLAY-DEC>
318 <OVERLAY-ELEMENT>
319 <OVERLAY-EXPRESSION>
320 <OVERLAY-STRING>
321 <PACKING-SPECIFIER>
322 <POINTER-ITEM-DESCRIPTION>
323 <POINTER-LITERAL>
324 <POINTER>
325 <PREFIX>
326 <PRESET-INDEX-SPECIFIER-HEAD>
327 <PRESET-INDEX-SPECIFIER>
328 <PRESET-VALUES-OPTION>
329 <PRIMARY>
330 <PROC-CALL-STMT>
331 <PROC-DEC>
332 <PROC-DEF>
333 <PROC-HEADING>
334 <PROC-MODULE>
335 <PROGRAM-BODY>
336 <PSEUDO-VARIABLE>
337 <REARRANGE-DIRECTIVE>
338 <REDUCIBLE-DIRECTIVE>
339 <REF-SPEC-CHOICE-LIST>
340 <REF-SPEC-CHOICE>
341 <REF-SPEC>
342 <RELATIONAL-EXPRESSION>
343 <RELATIONAL-OPERATOR>
344 <REPEATED-PRESET-VALUES-OPTION>
345 <REPETITION-LIST-HEAD>
346 <RESERVED-WORD>
347 <RETURN-STMT>
348 <ROUND-OR-TRUNCATE>
349 <S-OR-U>
350 <SAFE>
351 <SHIFT-DIRECTION>
352 <SHIFT-FUNCTION>
353 <SIGN-FUNCTION>
354 <SIMPLE-DEF>
355 <SIMPLE-REF>
356 <SIZE-FUNCTION>
357 <SIZE-TYPE>
358 <SKIP-DIRECTIVE>
359 <SPACER>
360 <SPECIFIED-ENTRY-SPECIFIER>
361 <SPECIFIED-ITEM-DESCRIPTION>
362 <SPECIFIED-TABLE-BODY>
363 <SPECIFIED-TABLE-ITEM-DEC>
```

364 <SPECIFIED-TABLE-OPTIONS-LIST>
365 <SPECIFIED-TABLE-OPTIONS>
366 <STARTING-BIT>
367 <STATUS-CONSTANT>
368 <STATUS-INVERSE-ARGUMENT>
369 <STATUS-INVERSE-FUNCTION>
370 <STATUS-ITEM-DESCRIPTION>
371 <STATUS-LIST>
372 <STATUS>
373 <STMT-LIST>
374 <STMT-NAME-DEC-HEAD>
375 <STMT-NAME-DEC>
376 <STMT>
377 <STOP-STMT>
378 <STRUCTURE-SPECIFIER>
379 <SUB-ATTRIBUTE>
380 <SUB-DEC>
381 <SUB-DEF-LIST>
382 <SUB-DEF>
383 <SUB-NAME>
384 <SUBSCRIPT>
385 <SYSTEM GOAL SYMBOL>
386 <TABLE-DEC>
387 <TABLE-DESCRIPTION>
388 <TABLE-PRESET-LIST>
389 <TABLE-PRESET>
390 <TABLE-TYPE-DEC>
391 <TABLE-TYPE-SPECIFIER>
392 <TARGET-LIST>
393 <TARGET>
394 <TERM>
395 <THEN-PHRASE>
396 <TRACE-DIRECTIVE>
397 <TYPE-DEC>
398 <TYPE-NAME>
399 <UNBALANCED-FOR-STMT>
400 <UNBALANCED-IF-STMT>
401 <UNBALANCED-STMT>
402 <UNBALANCED-WHILE-STMT>
403 <UNBALANCED>
404 <WHICH-BOUND>
405 <WHILE-CLAUSE>
406 <WHILE-PHRASE>
407 <WORDS-PER-ENTRY>
408 <XOR-FORMULA>

E. Appendix: Test J73 Program for Translation

```
START                    "Current 15 Oct 81"

PROGRAM TRANSLATOR;

BEGIN "translator"

  ITEM NEW'NAME C 3;
  ITEM TEST'FLAG B ;
  ITEM OLD'CHAR C;
  ITEM INTEGER'NUMBER U 3;
  ITEM FLOAT'NUM F;
  ITEM NEW'FLOAT'NUM F;
  ITEM NEW'INTEGER'NUMBER U;
  ITEM NEW'INTEGER'NUM    U;

HERE:

  FOR I:0 BY 1 WHILE I < INTEGER'NUMBER;
    BEGIN
    OLD'CHAR = 'B';
    FLOAT'NUM = .0;
    END

  INTEGER'NUMBER = 1234567890;
  FLOAT'NUM = 1234567890.1234567890;
  FLOAT'NUM = .1E-5;
  FLOAT'NUM = .1E+5;
  FLOAT'NUM = .1005;
  FLOAT'NUM = 1.12345E10;

  CASE INTEGER'NUMBER;

    BEGIN %case%

      (DEFAULT) : OLD'CHAR = 'A';
      (1)       : TEST'FLAG = TRUE;
      (2:3,5)   : NEW'FLOAT'NUM  = 20.1234567;

    END  "case"

  WHILE NEW'INTEGER'NUMBER = 3;
    TESTPROC;;

  IF INTEGER'NUMBER = 5;
     GOTO HERE;

  IF NEW'NAME = 'XYZ';
     NEW'NAME = 'YES';
```

```
  ELSE
     NEW'NAME = 'NO ';
"End of main program---start procedures"

PROC TESTPROC;
BEGIN "testproc"

  IF TEST'FLAG;
     BEGIN
       NEW'NAME = 'ABC';
       FLOAT'NUM = 55.01;
     END
END   "testproc"

END  "translator"
TERM
```

F. Appendix: Equivalent Ada Program

Note: The syntax for the Ada case in this example is incorrect due to the ordering error in the <WHEN OTHERS> alternative. Ada requires that this case alternative be in the last position when used.

```
  PROCEDURE TRANSLATOR IS
    NEW_NAME : STRING ( 1 .. 3 ) ;
    TEST_FLAG : BOOLEAN ;
    OLD_CHAR : STRING ( 1 .. 1 ) ;
    INTEGER_NUMBER : INTEGER RANGE 0 .. 2 ** ( 3 ) - 1 ;
    FLOAT_NUM : FLOAT ;
    NEW_FLOAT_NUM : FLOAT ;
    NEW_INTEGER_NUMBER : INTEGER RANGE 0 .. INTEGER'LAST ;
    NEW_INTEGER_NUM : INTEGER RANGE 0 .. INTEGER'LAST ;
    PROCEDURE TESTPROC IS
      BEGIN
      IF TEST_FLAG THEN
          NEW_NAME := "ABC" ;
          FLOAT_NUM := 55.01 ;
          END IF ;
        END ;
    BEGIN
    << HERE >>
 ----------------------------WARNING-----------------------------
 --                                                             --
 --    THE CODE BETWEEN THIS BRACKET AND THE FOLLOWING          --
 --          BRACKET MAY NOT BE FULLY TRANSLATED                --
 --                                                             --

      FOR I : 0 BY 1 WHILE I < INTEGER_NUMBER ;
      OLD_CHAR := "B" ;
      FLOAT_NUM := .0 ;


 --                                                             --
 --                                                             --
 --    THE CODE BETWEEN THIS BRACKET AND THE PREVIOUS           --
 --          BRACKET MAY NOT BE FULLY TRANSLATED                --
 --                                                             --
 ----------------------------WARNING-----------------------------
      INTEGER_NUMBER := 1234567890 ;
      FLOAT_NUM := 1234567890.1234567890 ;
      FLOAT_NUM := 0.1E-5 ;
      FLOAT_NUM := 0.1E+5 ;
      FLOAT_NUM := 0.1005 ;
```

```
      FLOAT_NUM := 1.12345E10 ;
      CASE INTEGER_NUMBER IS
        WHEN OTHERS => OLD_CHAR := "A" ;
        WHEN 1 => TEST_FLAG := TRUE ;
        WHEN 2 .. 3 | 5 => NEW_FLOAT_NUM := 20.1234567 ;
        END CASE ;
      WHILE NEW_INTEGER_NUMBER = 3 LOOP
      TESTPROC ;
        END LOOP ;
      NULL ;
      IF INTEGER_NUMBER = 5 THEN
        GOTO HERE ;
        END IF ;

--------------------------WARNING-----------------------------
--                                                          --
--    THE CODE BETWEEN THIS BRACKET AND THE FOLLOWING       --
--          BRACKET MAY NOT BE FULLY TRANSLATED             --
--                                                          --

      IF NEW_NAME = "XYZ" THEN
        NEW_NAME := "YES" ;
        ELSE


--                                                          --
--                                                          --
--    THE CODE BETWEEN THIS BRACKET AND THE PREVIOUS        --
--          BRACKET MAY NOT BE FULLY TRANSLATED             --
--                                                          --
--------------------------WARNING-----------------------------
        NEW_NAME := "NO " ;
        END IF ;
      END ;
```

Vita

Eric L. East was born in a farming region near Tyronza, Arkansas on March 2, 1948. He was graduated from Marked Tree High School in Marked Tree, Arkansas in May 1966. He attended Arkansas State College for a year and a half before enlisting in the United States Air Force in February, 1968. He worked in the computer support area in the Standard Base Level Supply System during his enlisted years. After a tour in Vietnam (1969-1970), he became a Univac 1050-II computer operator. During the next several years, he worked on his undergraduate degree in computer science in his spare time and graduated from Mississippi State University, Starkville, Mississippi in December 1975 with a Bachelor of Science Degree in Computer Science. After attending Officers Training School via the Airmens Education and Commissioning Program, he was commissioned a second lieutenant on April 26, 1976. In June, 1980 he entered the School of Engineering, Air Force Institute of Technology, to pursue a Masters of Science Degree in Computer Technology.

Permanent address: Box 655
Marked Tree, Arkansas
72365

Present address: 9559 Miller's Ridge
San Antonio, Texas
78239

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/MA/81D-2	2. GOVT ACCESSION NO. AD-A115 548	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) CONTINUED DEVELOPMENT OF THE JOVIAL (J73) TO ADA TRANSLATOR SYSTEM		5. TYPE OF REPORT & PERIOD COVERED MS THESIS 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Eric L. East Capt USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT/EN) Wright-Patterson AFB, OH 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Avionics Laboratory (AFWAL/AAAF-2) Wright-Patterson AFB, OH 45433		12. REPORT DATE December 1981 13. NUMBER OF PAGES 87
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/ DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

15 APR 1982

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB, OH 45433

18. SUPPLEMENTARY NOTES

Approved for public release; IAW AFR 190-17

F. C. Lynch, Major, USAF
Director of Information

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Language Translation
Grammars
Parsing
Embedded-computer Software
Programming Languages
JOVIAL (J73)
Ada

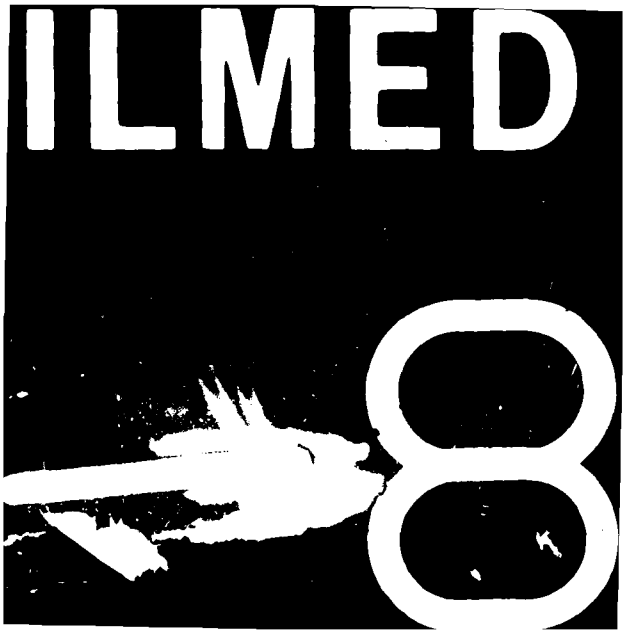
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Department of Defense recently announced Ada as the new standard high order programming language for embedded-computer software which will replace JOVIAL(J73), the current Air Force standard. Since the Air Force has a considerable investment in J73 software, there is a real need for a J73-to-Ada translator. The subject of this thesis is the continued development of this translator which was originally developed by Capt Richard L. Brozovic (AFIT/GCS/EE/80D-5).

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)


ILMED
8